Event Processor Host - imperative Ergänzung zu Azure Stream Analytics

Patrick Schubert und Daniel Wiese
Fakultät Informatik
Hochschule Furtwangen University, Deutschland
{p.schubert, daniel.wiese}@hs-furtwangen.de

***Zusammenfassung*—In dieser Arbeit wird der von Microsoft entwickelte `EventProcessorHost` vorgestellt und analysiert. Dazu wird erläutert, wie die Ereignisverarbeitung in Azure funktioniert und im Zuge dessen die zwei von Microsoft bereitgestellten Möglichkeiten des Stream Processings vorgestellt. Namentlich ist dies neben dem EventProcessorHost Azure Stream Analytics. Azure Stream Analytics verfolgt einen deklarativen Ansatz zur Verarbeitung von Ereignissen, während der EventProcessorHost einen imperativen Ansatz verfolgt. Die Vor- und Nachteile der jeweiligen Ansätze werden dabei als Bestandteil dieser Arbeit erläutert. Nach genauerer Untersuchung des EventProcessorHosts und Recherchen zu Stream Analytics, werden beide Ansätze direkt miteinander verglichen und aufgezeigt, warum der EventProcessorHost keine Alternative zu Stream Analytics ist. Ebenso wird anhand eines Vergleiches zu anderen Stream Processing-Frameworks gezeigt, dass es sich beim EventProcessorHost nicht um ein vollsändiges Stream Processing-Framework handelt. Jedoch eignet er sich als eine leichtgewichtige Ergänzung für komplexere Queries und kann mit Stream Analytics gemeinsam genutzt werden kann, ohne weitere Frameworks aufsetzen zu müssen.**

***Index Terms*—EPH, Event Processor Host, Azure, Stream Analytics, Streaming Systems**

I. Einleitung

Die Welt wird immer stärker vernetzt. Schon kleinste Geräte besitzen eine große Menge an Fähigkeiten. Um *smarte* Anwendungen zu ermögliche, werden sie mit anderen Geräten und dem Internet vernetzt und senden Informationen in Form von Ereignissen (Events) an Interessenten. Auf Basis dieser Ereignisse, werden in Echtzeit Analysen ausgeführt. Kombiniert man die Daten verschiedener Geräte und Sensoren, ermöglicht das, die Daten zu interpretieren, indem innerhalb der Datenmengen nach Mustern gesucht wird, die für das Einsatzgebiet von Bedeutung sind.

Da immer größeren Datenmengen produziert werden, werden auch immer mehr Streaming Systeme eingesetzt, um diese zu verarbeiten. Während 2018 noch 33 ZB an Daten produziert wurden, wird die Datenmenge für 2025 auf 175 ZB geschätzt [1]. Dieser Anstieg erfordert Wissen und den Einsatz verschiedener Technologien zum Bewältigen dieser Datenmenge. Vor allem die Anforderung, die Ergebnisse in Echtzeit zu erhalten und zu verarbeiten führt zu einer stark erhöhten Nachfrage nach Streaming Systemen [2].

Die weltweiten Umsätze für Big-Data- und Analytics-Software betrug 2018 weltweit rund 60 Milliarden US-Dollar [3]. Microsoft konnte in dem Zeitraum, den die Statistik betrachtet, den größten Umsatzzuwachs einfahren. Microsoft bietet Datenanalysen als Cloud-Lösung als Teil von *Azure* an. Im Rahmen dieser Arbeit betrachten wir zwei Angebote seitens Microsoft zur Verarbeitung von Streaming-Daten. Eins dieser Angebote ist Azure Stream Analytics (ASA) und das zweite Angebot ist der Event Processor Host (EPH).

II. Verwandte Arbeiten

Die ältesten Beiträge zum Thema ASA gehen bis in das Jahr 2015 zurück [4] [5]. Der erste Commit zum EPH fand im August 2016 statt [6]. Seither beschäftigten sich nur wenige wissenschaftliche Arbeiten mit dem Thema ASA oder EPH.

Basak et al. stellen in ihrem Buch „Stream Analytics with Microsoft Azure“ [2] einige Anwendungsfälle vor und zeigen mögliche Umsetzungen mit ASA auf. Allerdings wird in dem Buch nicht auf den EPH eingegangen.

Rob Tiffany hat Blog-Einträge zu ASA und dem EPH veröffentlicht. Als Beispiel verwendet er in seinen Blogs einen *Parkservice*. In Smart Cities soll es einem Fahrer vereinfacht werden, einen Parkplatz zu finden. Dieses Szenario stellt er mit ASA [5] und in einem weiteren Blog mit dem EPH vor [7]. Dabei geht er beim EPH vor allem auf die Anwendungsentwicklung ein und nicht auf einzelne Konzepte wie Fail-Over oder Load Balancing mit dem EPH.

Die Literatur zum EPH für unsere Arbeit geht kaum über Microsofts Dokumentation zum EPH hinaus. Diese Arbeit soll deshalb auch den bisher fehlender Überblick über die Fähigkeiten und Einschränkungen sowie die Funktionsweise des EPH bieten.

III. Event Processing in Azure

Microsoft gliedert die Architektur von Echtzeitverarbeitungssystemen in sechs Bestandteile [8] (siehe Abbildung 1). Zwei dieser sechs Bestandteile sind *Ingestion* und *Stream Processing*.

Ingestion: Für die Datenstromverarbeitung ist das Erfassen und Speichern von empfangenen Nachrichten erforderlich. Diese Nachrichten können von Konsumenten abgeholt und verarbeitet werden. Häufig werden hierfür Nachrichten-Broker eingesetzt, wie z.B. Azure Event Hubs.

Stream Processing: Mit Hilfe von Filtern, Aggregationen und weiterer Analysen werden die Nachrichten verarbeitet. Ermöglicht wird dies durch Systeme wie ASA und dem EPH.

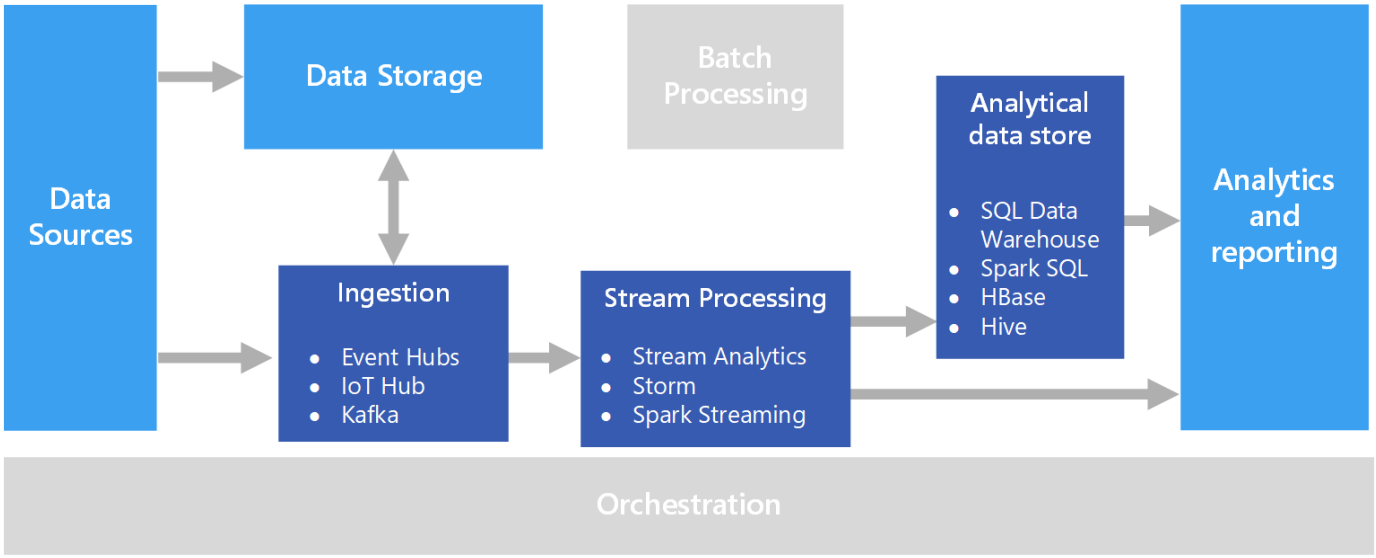

Abbildung 1. Real Time Processing Pipeline [8]

A. *Event Hub*

Azure Event Hubs sind eine von Microsoft angebotene Platform-as-a-Service (PaaS) zum Streamen von Millionen von Events. Das Nachrichtenstreaming wird durch ein partitioniertes Consumermuster realisiert [9].

Dazu ist der Event Hub in mehrere Partitionen unterteilt, die jeweils einen Teil des Event-Strom enthalten. Dabei verbleiben die Events auch nach dem Lesen durch einen Nachrichtenkonsument für einen definierten Zeitraum in den Partitionen. Jede Partition enthält also eine zeitlich geordnete Menge an Events, die ihr aus der Gesamtmenge aller Events zugeteilt worden sind. Durch lesen aus verschiedenen Partitionen kann die Arbeit auf mehrere Konsumenten verteilt werden, ohne dass ein zweiter Konsument dieselbe Nachricht ein weiteres Mal konsumiert. Ist die Last gering kann ein Konsument auch mehrere Partitionen übernehmen. Auf diesem Wege skaliert der Event Hub horizontal. [10]

Nachrichtenkonsumenten lassen sich zudem wie bei Apache Kafka in Konsumentengruppen einteilen. Jede Gruppe konsumiert dabei die Nachrichten jeder Partition (üblicherweise) genau einmal. [9]

Nachrichtenproduzenten können Ereignisse über Hypertext Transfer Protocol (HTTP) oder Advanced Message Queuing Protocol (AMQP) an den Event Hub senden. AMQP ist ein standardisiertes Framing- und Übertragungsprotokoll, das eine asynchrone, sichere und zuverlässige Nachrichtenübertragung ermöglicht [2].

B. *Azure Stream Analytics*

ASA ist eine cloudverwaltete Complex Event Processing (CEP) Lösung zum Analysieren von Streamingdaten [11] [2].

Es erlaubt die Filterung, Verarbeitung und Aggregation von Daten in Zeitfenstern. Dazu verwendet ASA eine deklarative SQL-artige Abfragesprache [2].

Eigene Funktionalität kann dabei über User Defined Functions (UDFs) in .NET oder JavaScript in ASA integriert werden. Für diese UDFs gelten jedoch Einschränkungen. UDFs sind laut Microsoft zustandslos und skalar. Mehrere Argumente oder Rückgabewerte sind nicht möglich. Sie sollten zudem idempotent, also mit dem gleichen Ergebnis wiederholbar sein. Bei UDFs in JavaScript wird zudem nur ein begrenzter Sprachumfang unterstützt. Der Zugriff auf externe HTTP-Endpunkte beispielsweise ist nicht möglich. Dadurch, dass die Funktionen zustandslos sind, sind auch keine benutzerdefinierten Aggregate möglich. Es gibt jedoch die Möglichkeit spezielle User Defined Aggregates (UDAs) in JavaScript zu erstellen [11].

Für eine flexiblere Datenverarbeitung wird explizit auf den EPH oder auch Spark Streaming verwiesen [11, p. 121].

Als Ein- und Ausgaben kann ASA eine Vielzahl an Azurediensten nutzen beispielsweise EventHubs, IoTHubs, BlobStorage, Datenbanken aber auch Azure Functions oder PowerBI. Externe Datenquellen außerhalb von Azure werden nicht unterstützt. Als Datenformat können standardmäßig nur JSON und AVRO verwendet werden. Benutzerdefinierte Serialisierer und Deserialisierer sind aber möglich [11].

ASA bietet bei der Nachrichtenverarbeitung und -auslieferung eine *exactly-once*-Garantie [11]. Abfragen können außerdem durch das Schlüsselwort `PARTITION BY` parallelisiert werden. Die Aufteilung der Abfrage muss dabei der Aufteilung der Nachrichten auf die Partitionen des EventHubs entsprechen, der als Nachrichtenquelle genutzt wird. Zudem muss die Anzahl der Ein- und Ausgabepartitionen identisch sein [11].

C. *Event Processor Host*

Der EPH ist ein intelligenter Konsument von Ereignissen. Intelligent bezeichnet in diesem Fall die Fähigkeit Leasings und Prüfpunkte zu verwalten sowie mit der Anzahl an Partitionen zu skalieren [12]. Er kann als Alternative [7] zu ASA dienen, hat aber einen gravierenden Unterschied. Die Ereignisverarbeitung wird im EPH wird nicht deklarativ beschrieben, sondern er führt vom Nutzer entwickelten imperativen Code aus. Der `EventProcessorHost` ist eine Klasse in den von

Microsoft bereitgestellten Software Development Kits (SDKs) für Azure Event Hubs [13].

Für jede Partition auf dem Event Hub, wird von einem EPH ein `EventProcessor` angelegt. Die Prozessoren können sich dabei über mehrere Instanzen (mehrere EPHs) verteilen, sofern mehrere EPHs existieren. [12] Somit können auf verschiedenen Systemen EPHs existieren, die insgesamt über alle Instanzen so viele Eventprozessoren haben, wie es Partitionen am Event Hub gibt. Fällt ein Host aus, vergrößert ein anderer die Anzahl seiner Prozessoren und übernimmt somit die des ausgefallenen Hosts [12].

Für diese Arbeit wird das .NET-SDK als Referenz-SDK verwendet. ([14]) Da es sich beim EPH um einen programmatischen Ansatz handelt, werden wir auch auf einzelne definierte Schnittstellen eingehen.

Im Wesentlichen besitzt der EPH, der in der Hauptklasse `EventProcessorHost` definiert wird, drei Fähigkeiten:

- `RegisterEventProcessorFactoryAsync` - Factory zum Erzeugen der Prozessoren registrieren
- `RegisterEventProcessorAsync<T>` - Prozessortyp registrieren
- `UnregisterEventProcessorAsync` - Stoppen des EPH

Die Klasse des `EventProcessorHost` ist als nicht ableitbare Klasse definiert. Hier ist somit keine eigene Implementierungsarbeit möglich und erforderlich. Die Events werden an eine Klasse übergeben, die das `IEventProcessor`-Interface implementiert. Dieser sogenannte Event-Prozessor ist selbst zu implementieren und enthält die erforderliche Geschäftslogik zur Ereignisverarbeitung. Das Interface deklariert vier Methoden [15]:

- `OpenAsync` - Prozessor wird initialisiert
- `CloseAsync` - Prozessor wird geschlossen
- `ProcessEventsAsync` - Verarbeitung der eingehenden Events (Liste von Events)
- `ProcessErrorAsync` - Wird aufgerufen, wenn die Verarbeitung einen Fehler wirft

In allen Methoden wird ein `PartitionContext` übergeben, in dem alle benötigten Informationen zu den Event Hub Partition sind, die der Event-Prozessor bearbeitet. Neben dem Besitzer und den Lease-Informationen befindet sich auch der Checkpoint im Partitionskontext. [16] Der Checkpoint markiert den Punkt in einer Partition, der von einem Event-Prozessor bereits bearbeitet wurde. Mit der Methode `CheckpointAsync` des Partitionskontextes wird der Checkpoint über den `CheckpointManager` aktualisiert und in die Cloud synchronisiert. Somit steht dem Entwickler frei, wann er eine Menge an Daten als verarbeitet betrachtet. Im Falle eines Ausfalls übernimmt ein anderer Prozessor, der am letzten gesetzten Checkpoint mit der Verarbeitung fortfährt. [12]

IV. Imperatives und Deklaratives Event Processing

Einer der größten Unterschiede zwischen ASA und dem EPH ist das Programmierparadigma. Während bei ASA eine deklarative Abfragesprache genutzt wird [11], wird beim EPH die Ereignisverarbeitung durch Programmcode imperativ beschrieben [12].

Deklarative Sprachen zeichnen sich dadurch aus, dass beschrieben wird, was verarbeitet werden soll und welches Ergebnis berechnet werden soll, ohne die Schritte zu beschreiben, die zu diesem Ergebnis führen. Dies ist nur möglich wenn die Vorgehensweise bereits beschrieben und als Funktion hinterlegt wurde. Deklarative Programmierung ist also nur möglich, wenn bereits Operationen durch imperative Programmierung definiert wurden [17]. Sie bietet den Vorteil einer hohen Wiederverwendung. Im System hinterlegte Operationen werden für viele verschiedene Abfragen immer wieder genutzt. Zudem werden die Details der Vorgehensweise versteckt. Dies führt zu deutlich prägnanteren und kürzeren Ausdrücken [17]. So kann ein Nutzer beispielsweise eine deklarative Sprache wie SQL verwenden, ohne sich Gedanken über die Internas des verwendeten Datenbanksystems zu machen. Auch eine automatische Lastverteilung und Skalierung für die Operationen kann bereits im System hinterlegt werden.

Das deklarative Vorgehen hat den Nachteil, dass der Nutzer auf die Funktionalität beschränkt ist, die bereits hinterlegt wurde. Der Sprachumfang ist begrenzt und es geht Flexibilität verloren. Deshalb versuchen Systeme wie ASA, Nutzern die Möglichkeit zugeben, in beschränktem Umfang eigene Funktionalität hinzuzufügen [11]. Ein Beispiel hierfür sind die bereits erwähnten UDFs, bei der durch imperative Beschreibung eine neue Funktion erstellt werden kann.

Bei der imperativen Vorgehensweise wird der genaue Ablauf Schritt für Schritt beschrieben [17]. Dazu können Skriptsprachen wie JavaScript oder auch Programmiersprachen .NET, Java oder C++ genutzt werden. Der Sprachumfang dieser Sprachen ist deutlich mächtiger, als der der meisten deklarativer Sprachen. Die genannten Sprachen werden deshalb auch als General Purpose Languages (GPL) bezeichnet, da sie nicht speziell auf ein Anwendungsfeld zugeschnitten sind. Der erstellte Code ist meist länger, dafür gibt es die Möglichkeit, bekannte Bibliotheken einzubinden und zu nutzen.

Um den Aufwand bei der Erstellung von Streaming Lösungen zu reduzieren, sollten da, wo es möglich ist, Frameworks oder Systeme verwendet werde, die bereits die benötigte Funktionalität besitzen, d.h. wenn es für die gewünschte Verarbeitung des Ereignisstroms bereits die nötige Funktionalität in ASA gibt, dann sollte es verwendet werden. Der EPH ermöglicht es bei Szenarien, die sich nicht mit dem Sprachumfang von ASA ausdrücken lassen und die sich auf nicht problemlos als UDF abbilden lassen, den exakten Verarbeitungsablauf in einer GPL imperativ zu beschreiben. Insbesondere die Interaktion mit externen Systemen, um beispielsweise die Ereignisse mit externen Daten anzureichern, lassen sich so zuverlässig realisieren. Auch komplexe Verarbeitungsschritte lassen sich möglicherweise einfacher imperativ beschreiben. Auch hier kann der EPH genutzt werden, falls die Einschränkung auf skalare UDFs bei ASA zu streng für das Anwendungsszenario ist.

V. Fail Over und Load Balancing im Event Processor Host

Wie bereits in Unterabschnitt III-C dargestellt, wird von Microsoft dokumentiert, dass es sich beim EPH um einen intelligenten Konsumenten handelt. Auch wird beschrieben, dass alle Hosts insgesamt genau so viele Event-Prozessoren haben, wie es Partitionen gibt und jeder dieser Prozessoren genau eine Partition bearbeitet. Leider konnten der Dokumentation keine Informationen entnommen werden, die den Verteilungsmechanismus auf einzelne Hosts beschreiben. Ebenso fehlt die Dokumentation zum Fail Over. Es bleibt also offen, wie sich die Hosts bei einem Ausfall verhalten.

Um die Mechanismen, die hinter dieser Intelligenz stehen, zu verstehen, führten wir eine Quellcode-Analyse durch. Dabei betrachteten wir zunächst nur die einzelnen Klassen und untersuchten, was ihre jeweilige Aufgabe ist und wie die Klassen im Zusammenhang stehen. (siehe Abbildung 2)

Das `IEventProcessor`-Interface sowie der `PartitionContext` wurden bereits in Unterabschnitt III-C vorgestellt und werden deshalb hier nicht erneut beschrieben.

EventProcessorHost: Der `EventProcessorHost` ist die zentrale Schnittstelle um den Host zu starten und zu beenden. Er nimmt den zu instanziierenden `IEventProcessor` oder eine `IEventProcessorFactory`, die die Prozessoren erzeugen kann, entgegen.

PartitionPump: In der `PartitionPump` wird der tatsächliche `IEventProcessor` erzeugt. Sie liest die Events, aus der ihr zugehörigen Partition und leitet sie an den erzeugten Prozessor weiter.

PartitionManager: Der Partitionsmanager verwaltet alle Pumpen in einer Map und erzeugt diese auch. Anhand des Partition-Schlüssels in der Map ist die zugehörige PartitionPump einzelner Partitionen auffindbar. Über den `ICheckpointManager` und den `ILeaseManager` werden die Checkpoints und Leases persistiert. Die Standardimplementierung, die sich bereits im SDK befindet, nutzt dafür ein Azure Blob Storage. Dazu wird für jede Partition ein Blob in dem Azure Storage angelegt. Der Pfad setzt sich dabei aus dem Namen der Konsumentengruppe und der Partitions-ID zusammen:

Storage://<consumer-group-name>/<partition-id>

Das Leasing wird mit Hilfe des AzureStorage-SDK vorgenommen und Zugriffsbeschränkungen, die für die Parallelität erforderlich sind, werden durch das Storage abgedeckt. Auf den einzelnen Blobs der Partitionen werden Leasings (Locks) durchgeführt. Die Leasing-Dauer kann unendlich sein oder zwischen 15 und 60 Sekunden betragen [18]. Ist die Persistierung auf einem anderem Medium gewünscht, kann ein eigener Manager diese Interfaces implementieren [19], [20].

Im Partitionsmanager steckt die von Microsoft angesprochene Intelligenz. In dieser Klasse findet man den Fail Over- und Load Balancing-Mechanismus. Wird der Host gestartet, wird der Partitionsmanager in eine Endlosschleife versetzt (`RunLoopAsync(CancellationToken)`). Die Schleife wird über eine `Stopwatch` immer wieder getriggert, bevor die Leasing-Zeiten ablaufen. Das Erneuerungsintervall beträgt standardmäßig 10 Sekunden und die Leihdauer 30 Sekunden.

Der genaue Ablauf kann im untenstehenden Pseudo-Code nachvollzogen werden:

Listing 1. `PartitionManager::RunLoopAsync` Pseudo Code

```
1  allLeases = leaseManager.getAllLeases()
2  For lease in allLeases:
3      if lease.isOwnLease:
4          lease.renew()
5      else if lease.isExpired():
6          othersLeases.add(lease)
7  For lease in othersLeases:
8      if lease.isExpired():
9          leaseManager.acquire(lease)
10 if biggestOwner.numberLeases > myLeases + 2:
11     steal = allLeases.First(l => l.owner ==
           biggestOwner)
12     leaseManager.acquire(steal)
13 renewPumps()
```

Fail Over: Ein Fail Over wird im Kontext des EPH über den Ablauf des Locks auf einem Blob im Azure Storage getriggert. In jedem Durchlauf des Partitionsmanagers werden die Locks im Storage erneuert. Kommt es zum Absturz eines Hosts, kann dieser seinen Lock nicht erneuern und er wird automatisch von einem andern Host übernommen. Mit den Standardintervallen kann es demnach zu einer Verzögerung von 20 Sekunden kommen, wenn ein Host ausfällt. Die 20 Sekunden ergeben sich aus diesem möglichen Ablauf mit zwei Hosts:

Zeit in ms	Ereignis
0	Host A prüfte und erneuerte Leases
1	Host B prüfte und erneuerte Leases
2	Host B stürzt ab
10000	Host A prüfte und erneuerte Leases
10001	Leases von Host B laufen ab
20000	Host A prüfte und erneuerte Leases

Load Balancing: Das Load Blancing erfolgt durch eine primitiven Gleichverteilung der Partitionen auf die einzelnen Hosts. Bei jedem Durchlauf wird geprüft, ob ein anderer Host Leases für mehr als zwei Partitionen mehr besitzt. Ist dies der Fall, wird eine Partition dieses Hosts durch einen anderen übernommen. Ein tatsächlicher Lastenausgleich in Bezug auf die Systemressourcen wird nicht vorgenommen. Hat das Gesamtsystem viele Partitionen (max. 32 ohne Sonderabsprache mit Microsoft [21]) kann dieser Vorgang einige Zeit in Anspruch nehmen. Dies ist der Tatsache geschuldet, dass maximal eine Partition je Schleifendurchlauf im Partitionsmanager geklaut wird. Somit dauert es mit den Standardintervallen und 32 Partitionen insgesamt 150 Sekunden bis ein System mit 2 Hosts in einem ausgeglichenen Zustand ist.

At-Least-Once: Microsoft gibt für den EPH eine *at least once*-Garantie, dass eine Nachricht erhalten und verarbeitet wird [12]. Zum einen garantiert der EventHub nur eine *at least once*-Garantie bei der Auslieferung [22] und zum anderen kann innerhalb des EPH nicht sichergestellt werden, dass eine Event-Verarbeitung quittiert wurde. Eine Mehrfachverarbeitung kann dabei z.B. durch einen Ausfall eines Hosts herbeigeführt werden, wenn dieser die Nachricht zwar verarbeitet, aber noch keinen Checkpoint gesetzt hat. Auch ist es möglich, dass einem Host eine Partition entzogen wird bevor er die Verarbeitung im Checkpoint quittieren kann.

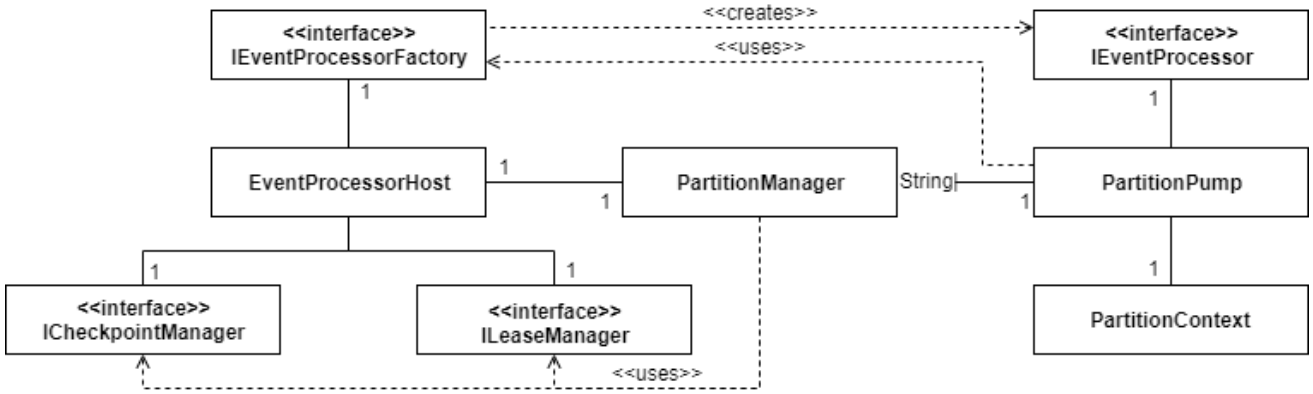

Abbildung 2. Architekturausschnitt zum `EventProcessorHost`

VI. Einordnung des Event Processor Hosts gegenüber anderen imperativen Frameworks

Wie in den letzten Abschnitten gezeigt werden konnte, besitzt der EPH einfache aber effektive Mechanismen zum Load Balancing und Fail Over. Er ermöglicht damit eine verteilte Ereignisverarbeitung mit benutzerdefiniertem Code. Im Vergleich zu anderen Lösungen bietet er darüber hinaus jedoch nicht die Möglichkeit, Ereignisse in Zeitfenstern zu gruppieren oder einen Zustand über mehrere Prozessor-Instanzen hinweg zu synchronisieren.

Das ermöglicht eine Abgrenzung gegenüber anderen Event-Processing-Frameworks, wie Apache Beam und Apache Samza.

Apache Beam ist ein High-Level Framework, das versucht, verschiedene andere Event-Processing-Frameworks wie Spark oder Samza zu abstrahieren [23]. Es verfolgt also einen fundamental anderen Ansatz als Microsofts EPH, bei dem nur grundlegende Funktionen zur Verfügung gestellt werden.

Interessanter für die Einordnung ist die Betrachtung der zugrundeliegenden Frameworks wie z.B. Apache Samza. Samza ist ein Event-Processing-Framework, das neben der Ausführung in YARN Clustern auch dafür designt ist, in andere Anwendungen integriert zu werden [24], [25]. Im Gegensatz zum EPH verfügt Samza auch über eine High-Level-API zur Ereignisverarbeitung. Entscheidender für den Vergleich ist jedoch, dass Samza eine Low-Level-API besitzt, die ebenso wie der EPH eine freie Ereignisverarbeitung mit benutzerdefiniertem Code erlaubt. Die Schnittstelle für StreamTasks ist dabei vergleichbar mit der des EPH. Der Nutzer implementiert eine Funktion, die die eingehenden Nachrichten übergeben bekommt und kann sie dann frei verarbeiten [26]. Auch Samza ist fehlertolerant, erlaubt Checkpoints sowie Load Balancing und darüber hinaus auch die beim EPH fehlende Synchronisation eines Zustandes bei Migration eines Tasks [25]. Dies ist notwendig, um z.B. eine Aggregation von Daten durch den benutzerdefinierten Code zu ermöglichen.

Der EPH ist also kein vollständiges Event-Processing-Framework im klassischen Sinne, sondern stellt nur rudimentäre Grundlage für eigene verteilte Lösungen zur Ereignisverarbeitung zur Verfügung. Mögliche Einsatzgebiete sind dabei das Anreichern von Nachrichten mit externen Informationen und Interaktionen mit Drittsystemen auf Basis von Ereignissen. Dabei ist es sinnvoll EPH zusammen mit anderen Lösungen wie ASA zu nutzen. Andere Frameworks, insbesondere Apache Samza bieten jedoch eine ähnliche Flexibilität und sind deutlich leistungsfähiger.

VII. Einsatz des Event Processor Hosts

Um den Nutzen des EPH in einem praktisches Szenario aufzuzeigen, wurde eine Lösung erstellt, die Änderungen an Wikipedia-Artikeln überwacht und versucht, Edit-Wars zu erkennen. Dabei wird der EPH genutzt, um die Ereignisse aus dem Wikipedia-Event-Stream mit weiteren externen Informationen über die Artikel anzureichern.

Der Aufbau dieser Lösung ist in Abbildung 3 dargestellt. Dabei werden die Ereignisse zuerst mittels einer Client Anwendung über die Wikipedia API gelesen und an einen Event Hub weitergeleitet, um sie innerhalb der Azure-Lösung nutzen zu können. Die Ereignisse werden anschließend mit Hilfe von ASA vorverarbeitet. Hierfür werden relevante Ereignisse, bei denen zwei Nutzer innerhalb des Zeitfensters von einem Tag am selben Artikel gearbeitet haben, zu Paaren aggregiert. Die gebildeten Paare werden dann mittels einer Anwendung auf Basis des EPHs klassifiziert und die Ergebnisse für den Nutzer in eine Datenbank abgelegt.

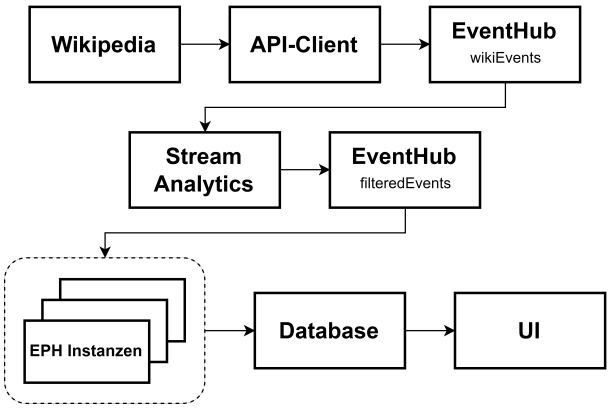

Abbildung 3. Einsatz des EPH zur Erkennung von Edit-Wars

Ziel dieser Lösung ist es, die Vorteile von ASA mit denen vom EPH zu kombinieren. ASA wird für die Aggregation über ein Zeitfenster genutzt, die im EPH nicht möglich ist, da

kein Zustand im `EventProcessor` verwaltet werden kann. Anschließend wird die Flexibilität des EPH genutzt, um benutzerdefinierten Analysecode auszuführen. Mit diesem Code werden die Diffs und die Revisionshistorie von den Wikipedia-Servern heruntergeladen, die als Basis für die Klassifikation genutzt werden.

VIII. Fazit

Mit dieser Arbeit konnten wir zeigen, was sich hinter dem von Microsoft bereitgestellten EPH verbirgt. Während Rob Tiffany den EPH als *Alternative* zu ASA bezeichnet, konnten wir viele Punkte nennen, die diesen nicht als echte Alternative erscheinen lässt.

Der erste große Unterschied, der den EPH von ASA unterscheidet ist der imperative Weg, der beim EPH erforderlich ist. Hier hat ASA bei weniger komplexen Aufgaben ganz klar einen Vorteil, was Lesbarkeit und Wartbarkeit betrifft. Bei komplexeren Aufgaben hingegen, können die ASA-Queries durchaus zu komplex werden und den Vorteil somit zunichte machen.

Man könnte annehmen, dass der EPH genau für dieses Szenario der komplexeren Queries entwickelt wurde, doch das stimmt auch nicht ganz. Dem EPH fehlen einige Konzepte des klassischen Stream Processings. So zum Beispiel unterstützt er keine Zeitfenster und insbesondere auch nicht einen Status z.B. für die Datenaggregation über Abstürze hinweg zu verwalten. All dies müsste durch die Einbindung weiterer Bibliotheken oder großem Eigenaufwand in den EPH eingebunden werden. Doch all dies scheint auch nicht der geplante Einsatzzweck des EPHs zu sein, denn Microsoft verweist bei komplexeren Szenarien ebenfalls auf andere Frameworks wie Apache Spark. Vielmehr ist er ein Leichtgewicht, das problemlos auf verschiedenen Systemen gestartet werden kann und einfache Aufgaben zuverlässig und schnell erledigt.

Der EPH ist also keinesfalls als Alternative zu ASA zu betrachten und soll dies auch gar nicht sein. Ebenso ist er auch kein vollwertiges Stream Processing-Framework. Vielmehr ist der EPH ein Werkzeug des Microsoft-Universums, mit dem eine Processing-Kette vereinfacht werden kann. Er ist leichtgewichtig und kann Queries, die mit ASA allein sehr komplex würden, vereinfachen. Somit kann man verhindern, dass große Frameworks wie Apache Samza eingesetzt werden müssen oder die ASA-Queries, die sowieso nur bedingt parallelisierbar sind, zu komplex werden.

Literatur

[1] D. Reinsel, J. Gantz, and J. Rydning, "The Digitization of the World – From Edge to Core," 2018, [Online] verfügbar unter: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, zuletzt besucht am: 16.12.2019.

[2] A. Basak, K. Venkataraman, R. Murphy, and M. Singh, *Stream Analytics with Microsoft Azure*. Birmingham: Packt Publishing, 2017.

[3] SAS Institute, "Umsatz Mit Big-data- Und Analytics-software Weltweit Von 2016 Bis 2018 Nach Anbieter (In Millionen Us-dollar," 2019, [Online] verfügbar unter: https://de.statista.com/statistik/daten/studie/780780/umfrage/umsatz-mit-big-data-und-analytics-software-weltweit/, zuletzt besucht am: 16.12.2019.

[4] Robert Sheldon, "Microsoft Azure Stream Analytics," 2015, [Online] verfügbar unter: https://www.red-gate.com/simple-talk/cloud/cloud-data/microsoft-azure-stream-analytics/, zuletzt besucht am: 20.12.2019.

[5] R. Tiffany, "Getting Started with Azure IoT services: Stream Analytics," 2015, [Online] verfügbar unter: https://www.linkedin.com/pulse/getting-started-azure-iot-services-stream-analytics-rob-tiffany, zuletzt besucht am: 16.12.2019.

[6] dlstucki and sjkwak, ".NET Core 1.0 (RTM) EventHubClient and EventProcessorHost (#210)," 2016, [Online] verfügbar unter: https://github.com/Azure/azure-event-hubs-dotnet/commit/52fb8bd0a75513b919615b7daf014054b3942e02, zuletzt besucht am: 20.12.2019.

[7] R. Tiffany, "Getting Started with Azure IoT services: Event Processor Host," 2015, [Online] verfügbar unter: https://www.linkedin.com/pulse/getting-started-azure-iot-services-event-processor-host-rob-tiffany, zuletzt besucht am: 16.12.2019.

[8] Microsoft Corporation, "Real time processing," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/real-time-processing, zuletzt besucht am: 16.12.2019.

[9] ——, "Azure Event Hubs — A big data streaming platform and event ingestion service," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-about, zuletzt besucht am: 16.12.2019.

[10] ——, "Scaling with Event Hubs," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-scalability, zuletzt besucht am: 16.12.2019.

[11] ——, "Übersicht über Azure Stream Analytics," 2019, [Online] verfügbar unter: https://docs.microsoft.com/de-de/azure/opbuildpdf/stream-analytics/TOC.pdf?branch=live, zuletzt besucht am: 20.12.2019.

[12] ——, "Event processor host," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-event-processor-host, zuletzt besucht am: 16.12.2019.

[13] ——, "Microsoft Azure Event Hubs," 2018, [Online] verfügbar unter: https://github.com/Azure/azure-event-hubs, zuletzt besucht am: 16.12.2019.

[14] ——, "Microsoft Azure Event Hubs Client for .NET," 2018, [Online] verfügbar unter: https://github.com/Azure/azure-event-hubs-dotnet, zuletzt besucht am: 16.12.2019.

[15] ——, "IEventProcessor Interface," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.eventhubs.processor.ieventprocessor, zuletzt besucht am: 17.12.2019.

[16] ——, "PartitionContext Class," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.eventhubs.processor.partitioncontext, zuletzt besucht am: 17.12.2019.

[17] T. McGinnis, "Imperative vs Declarative Programming," 2019, [Online] verfügbar unter: https://tylermcginnis.com/imperative-vs-declarative-programming/, zuletzt besucht am: 20.12.2019. [Online]. Available: https://tylermcginnis.com/imperative-vs-declarative-programming/

[18] Microsoft Corporation, "CloudBlob.AcquireLeaseAsync Method," 2019, [Online] verfügbar unter: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.storage.blob.cloudblob.acquireleaseasync, zuletzt besucht am: 17.12.2019.

[19] ——, "ILeaseManager Interface," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.eventhubs.processor.ileasemanager, zuletzt besucht am: 17.12.2019.

[20] ——, "ICheckpointManager Interface," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.eventhubs.processor.icheckpointmanager, zuletzt besucht am: 17.12.2019.

[21] ——, "Features and terminology in Azure Event Hubs," 2018, [Online] verfügbar unter: https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features, zuletzt besucht am: 17.12.2019.

[22] ——, "https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services," 2019, [Online] verfügbar unter: https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services, zuletzt besucht am: 17.12.2019.

[23] Apache Beam, "Apache Beam Programming Guide," 2019, [Online] verfügbar unter: https://beam.apache.org/documentation/programming-guide/, zuletzt besucht am: 18.12.2019.

[24] Apache Samza, "Samza - Background," 2019, [Online] verfügbar unter: http://samza.apache.org/learn/documentation/latest/introduction/background.html, zuletzt besucht am: 18.12.2019.

[25] ——, "Samza - Core concepts," 2019, [Online] verfügbar unter: http://samza.apache.org/learn/documentation/latest/core-concepts/core-concepts, zuletzt besucht am: 18.12.2019.

[26] ——, "Samza - Low level Task API," 2019, [Online] verfügbar unter: http://samza.apache.org/learn/documentation/latest/api/low-level-api.html, zuletzt besucht am: 18.12.2019.